Annotation-based property checking for systems software

Thomas Ball
Microsoft Research
tball@microsoft.com

Brian Hackett
Stanford University
bhackett@stanford.edu

Shuvendu K. Lahiri, Shaz Qadeer
Microsoft Research
{shuvendu,qadeer}@microsoft.com

Abstract

Specifying procedure contracts for buffer overruns in systems code has been effective in detecting and fixing thousands of vulnerabilities in Windows. However, providing annotations for checking more system-specific properties can be challenging for systems code, typically written in C, due to the presence of objects in the heap and pointer arithmetic. In this work, we present an annotation-based property checker for such programs. Our checker retains precision, scalability, automation, and high coverage, in spite of the expressiveness of the supported annotations. We demonstrate the feasibility of our approach by applying it to check properties related to the synchronization protocol on a core Microsoft Windows component with more than 300,000 lines of code and 1500 procedures. The tool found 45 serious bugs in the component with a modest annotation effort. Most of these bugs have since been fixed by the developers of the module.

1 Introduction

Developing and maintaining systems software such as operating systems kernel and device drivers is a challenging task. They consist of modules often exceeding several hundred thousand to millions of lines of code written in low-level languages such as C and C++. In many cases, these modules evolve over several decades and the original architects or developers may have long ago departed. Such software may become fragile through the accumulation of new features, performance tuning and bug fixes, often added in an ad-hoc manner. Given the astronomical number of paths in any real program, testing can only cover a relatively very small fraction of the paths in a module. Bugs found in the field often occur in these rarely exercised paths.

Static compile-time tools have the potential to discover bugs early in the development cycle with high coverage, and offer an attractive complement to testing. Microsoft's Standard Annotation Language (SAL) is a meta-level language for assisting static analysis tools to find bugs primarily related to buffer-overruns in C or C++ code [16]. It provides a set of annotations to describe how a function uses its parameters — the assumptions it makes about them, and the guarantees it provides on exit. Modular checking of these contracts, where each procedure is analyzed only with the contracts of other procedures, provides a scalable method for applying static analysis to millions of lines of code.

SAL annotations and static checking tools based on these annotations (such as ESPX [16]) have had a significant impact on the development process inside Microsoft. A substantial fraction of Windows source has been annotated with various SAL annotations and thousands of potential security vulnerabilities have been discovered and fixed. The annotations also serve as valuable documentation for the procedures, providing quick feedback of any breaking changes on the developer's desktop and preventing many severe bugs.

However, SAL-based checkers are custom built for a class of properties (e.g. buffer overrun checking in ESPX) and use ad-hoc techniques that sacrifice soundness for fewer false alarms. As a result, it is difficult to extend the checkers to support a large class of user-defined properties. Examples of such properties can range from generic properties such as "lock `l` is acquired and released alternately" and "field `f` in object `x` is protected by the lock `l`" to more system-specific properties such as the absence of *teardown races*[1], which requires knowledge of the synchronization protocol of the system. The ability to check such properties enables early diagnosis of bugs that are notoriously hard to detect and reproduce.

In this work, we present a general-purpose annotation-based property checker for systems software written in C, implemented in the HAVOC tool [6]. HAVOC provides an expressive annotation language to specify user-defined properties, and most intermediate assertions required to prove such properties. In spite of its expressiveness, the checker retains precision, scalability, automation and high coverage by a combination of the following:

1. An accurate memory model for C programs accounting for low-level operations such as pointer arithmetic

[1] Deletion or freeing of an object being accessed by another thread.

and addresses of internal fields.

2. Precise reasoning about call-free and loop-free fragments of code inside a procedure using the accurate memory model and the BOOGIE [5] verifier.

3. Scalable checking of constraints using state-of-the-art Satisfiability Modulo Theories (SMT) solver Z3 [10].

4. An annotation assistant for interprocedural inference of simple contracts based on a variant of the Houdini algorithm [14].

5. Prudent use of *explicit assumptions* about type-safety and type invariants about data structure shapes. These assumptions are required to verify high-level properties with low false alarm, but verifying them might require significant annotation overhead.

We demonstrate the feasibility of the approach by applying it on a core component COMP of the Windows kernel — the name of the module and the code fragments have been modified for proprietary reasons. The code base has more than 300,000 lines of C code and has evolved over two decades. The module has over 1500 procedures, with some of the procedures being a few thousand lines long — a result of the various feature additions over successive versions.

For this component, we specified and checked properties related to the synchronization protocol on its main heap allocated data structures. The properties include checking for resource leaks, improper lock usage, data races and teardown races. The highlights of the effort were:

1. HAVOC found 45 bugs in the COMP module that were confirmed by the developers and many of them have been fixed at the time of writing. Most of these bugs appear along error recovery paths indicating the mature and well-tested nature of the code and signifying the ability of HAVOC to detect subtle corner cases.

2. The checking required modest annotation effort of about 250 annotations for specifying the properties and operating system model, 600 annotations for procedure contracts. An annotation assistant generated around 3000 simple annotations, a bulk of the required annotation effort, to relieve the need for annotating such a large code base. This corresponds to roughly one manual annotation per 500 lines of code, or one per 2.5 procedures.

3. The tool currently reports 125 warnings, including the 45 confirmed bugs, when the checker runs on the annotated code base. The extra warnings can be minimized at the cost of more annotations.

The rest of the paper is organized as follows: In Section 2, we illustrate our approach with a simplified problem of data race checking from COMP; we describe our annotations and the checker in Section 3 and the annotation assistant in Section 4. We describe the details of our case study on COMP in Section 5, discuss related work in Section 6, and finally conclude in Section 7.

2 Illustrative example

In this section, we illustrate the challenges involved in checking properties of low-level systems software, with an example of data-race checking on the main heap-allocated data structures in COMP. These data structures are fairly representative of low-level systems software.

Our example demonstrates that modular verification of data-race freedom on COMP code requires (i) annotations involving pointer arithmetic, aliasing, and conditional specifications, (ii) loop invariants to deal with unbounded sources of data (e.g., lists), and (iii) explicitly assumed type invariants for trading off soundness for scalability.

We first describe high-level details of the data structure and the synchronization protocol, some procedures manipulating these structures, and finally the annotations to check the absence of data races.

2.1 Example

Figure 1 describes a few types for the heap-allocated data structures in COMP. The type `LIST_ENTRY` is the generic type for (circular) doubly-linked lists in most of Windows source code. It contains two fields `Flink` and `Blink` to obtain the *forward* and *backward* successors of a `LIST_ENTRY` node respectively in a linked list. An object of type `NODEA` contains a list of children objects of type `NODEB` using the field `NodeBQueue`. Figure 3 describes the shape of the children list for any `NODEA` object. Each child `NODEB` node also maintains a pointers to its *parent* `NODEA` object with the `ParentA` field.

The macro `CONTAINING_RECORD` (defined in Figure 1) takes a pointer `addr` to an internal field `field` of a structure of type `type` and returns the pointer to the enclosing structure by performing pointer arithmetic. The helper macros `ENCL_NODEA` and `ENCL_NODEB` uses the `CONTAINING_RECORD` macro to obtain pointers to enclosing `NODEA` and `NODEB` structures respectively, given a pointer to their respective `LIST_ENTRY` fields. The `CONTAINING_RECORD` macro is frequently used in systems software and is a primary source of pointer arithmetic.

Since these objects can be accessed from multiple threads, one needs a synchronization mechanism to ensure the absence of data-races on the fields of these objects. Each `NODEA` structure maintains a field `Resource`, which

```
typedef struct _LIST_ENTRY{
  struct _LIST_ENTRY *Flink, *Blink;
} LIST_ENTRY, *PLIST_ENTRY;

typedef struct _NODEA{
  PERESOURCE Resource;
  LIST_ENTRY NodeBQueue;
  ...
} NODEA, *PNODEA;

typedef struct _NODEB{
  PNODEA     ParentA;
  ULONG State;
  LIST_ENTRY NodeALinks;
  ...
} NODEB, *PNODEB;

#define CONTAINING_RECORD(addr, type, field)\
   ((type *)((PCHAR)(addr) -\
             (PCHAR)(&((type *)0)->field)))

//helper macros
#define ENCL_NODEA(x)\
        CONTAINING_RECORD(x, NODEA, NodeBQueue)
#define ENCL_NODEB(x)\
        CONTAINING_RECORD(x, NODEB, NodeALinks)
```

Figure 1. Data structures and macros used in the example.

is a pointer to a ERESOURCE structure that implements a reader-writer lock. The lock not only protects accesses to the fields in the NODEA structure but additionally also protects the NodeALinks, ParentA and State fields in any of its NODEB children.

Figure 2 describes three procedures that manipulate the NODEA and NODEB objects. For this example, we indicate the lines containing an annotation using an empty comment "/**/" at the start of the line. The procedure CompClearChildState takes a NODEA object NodeA and clears a mask StateMask from the State field of any NODEB child that matches this mask. It uses the procedure CompFindChildState in a loop to find all the children that match the StateMask and then calls CompClearState on the child to actually clear the mask. The procedure CompFindChildState iterates over the children for a NODEA object and returns either the first child that matches the mask, or NULL if no such child exists.

To encode the data race freedom property on the fields of NODEA and NODEB objects, we introduce assertions that each access (read or write) to a field is guarded by the Resource lock in the appropriate NODEA object. The three procedures clearly satisfy data race freedom since the lock on the NODEA object is acquired by a call to CompAcquireNodeAExcl inside CompClearChildState before any of the operations.

```
#define __resrA_held(x)\
        __resource(''NODEA_RESOURCE'',x) > 0

VOID CompClearChildState(PNODEA NodeA,
                         ULONG StateMask) {

  CompAcquireNodeAExcl(NodeA);
  PNODEB NodeB;
  CompFindChildState(NodeA, StateMask, &NodeB);

  /**/__loop_invariant(NodeB != NULL ==>
  /**/                 NodeB->ParentA == NodeA)
  while (NodeB != NULL) {
     CompClearState(NodeB, StateMask);
     CompFindChildState(NodeA, StateMask, &NodeB);
  }
  CompReleaseNodeA(NodeA);
}

/**/__requires(__resrA_held(NodeA))
/**/__ensures (*PNodeB != NULL ==>
/**/           (*PNodeB)->ParentA == NodeA)
VOID CompFindChildState(PNODEA NodeA,
                        ULONG StateMask,
                        PNODEB* PNodeB) {
  PLIST_ENTRY Entry = NodeA->NodeBQueue.Flink;

  /**/__loop_invariant(
  /**/  Entry != &NodeA->NodeBQueue ==>
  /**/  ENCL_NODEB(Entry)->ParentA == NodeA
  /**/  )
  while (Entry != &NodeA->NodeBQueue) {
     PNODEB NodeB = ENCL_NODEB(Entry);
     if (NodeB->State & StateMask != 0) {
         *PNodeB = NodeB; return;
     }
     Entry = Entry->FLink;
  }
  *PNodeB = NULL; return;
}

/**/__requires(__resrA_held(NodeB->ParentA))
VOID CompClearState(PNODEB NodeB,
                    ULONG StateMask) {
   NodeB->State &= ~StateMask;
}
```

Figure 2. Procedures and annotations for data race freedom.

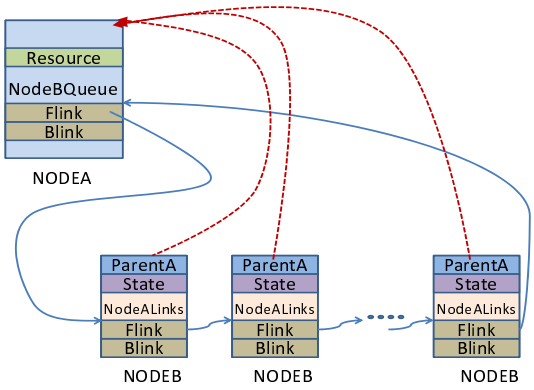

Figure 3. The list of `NODEB` children of a `NODEA`.

```
#define FIRST_CHILD(x)   x->NodeBQueue.Flink
#define NEXT_NODE(x)     x->NodeALinks.Flink

__type_invariant(PNODEA x){
    ENCL_NODEA(FIRST_CHILD(x)) != x  ==>
    ENCL_NODEB(FIRST_CHILD(x))->ParentA == x
)

__type_invariant(PNODEB y){
    NEXT_NODE(y) != &(y->ParentA->NodeBQueue) ==>
    y->ParentA == ENCL_NODEB(NEXT_NODE(y))->ParentA
)
```

Figure 4. Type invariants for `NODEA` and `NODEB` types.

2.2 Annotations

Now, let us look at the annotations required by HAVOC to verify the absence of the data race in the program.

The procedure `CompClearState` has a *precondition* (a condition inside `__requires` that holds at the entry to a procedure) that the `Resource` field of the `NodeB->ParentA` is held at entry; this ensures that the access to `NodeB->State` is property protected. The annotation `__resource (''NODEA_RESOURCE'', x)`, which the macro `__resrA_held(x)` expands to, returns the value of a *ghost field* called `''NODEA_RESOURCE''` inside `x`. The integer valued ghost field `''NODEA_RESOURCE''` tracks the state of the `Resource` lock in a `NODEA` object — a positive value denotes that the `Resource` is acquired. For brevity, we skip the annotations for `CompAcquireNodeAExcl` and `CompReleaseNodeA`, which increment and decrement the value of the ghost field, respectively.

The procedure `CompFindChildState` has a similar precondition on the `NodeA` parameter. The procedure also has a *postcondition* (a condition inside an `__ensures` that holds at exit) that captures child-parent relationship between the out parameter `PNodeB` and `NodeA`.

Let us inspect the annotations on `CompClearChildState`. We need a *loop invariant* (a condition inside `__loop_invariant` that holds every time control reaches the loop head) to ensure the precondition of `CompClearState` inside the loop. The postcondition of `CompFindChildState` ensures that the loop invariant holds at the entry of the loop and also preserved by any iteration of the loop.

Finally, consider the loop invariant in procedure `CompFindChildState`: the loop invariant is required for both proving the postcondition of the procedure, as well as to prove the absence of a data race on `NodeB->State` inside the loop. This loop invariant does not follow directly from the annotations on the procedure and the loop body. To prove this loop invariant, we specify two *type invariants* for `NODEA` and `NODEB` objects using the `__type_invariant` annotation in Figure 4. The type invariant on any `NODEA` object `x` states that if the children list of `x` is non-empty then the parent field `ParentA` of the first child points back to `x`. The type invariant for any `NODEB` object `y` states that if the next object in the list is not the head of the circular list, then the next `NODEB` object in the list has the same parent as `y`. The two type invariants capture important shape information of the data structures and together imply that all the `NODEB` objects in the children list of `NodeA` point to `NodeA`.

Proving type invariants require reasoning with quantifiers, which is undecidable, and may require significant annotation overhead. Hence, these type invariants are currently *assumed* (but not asserted) at various control locations in the program. The type invariants have to be written down only once for each type and therefore the complexity of writing these complex annotations is justified when checking a large code base such as COMP. Moreover, they can later be verified at the cost of more human effort.

3 Annotations and checker

In this section, we illustrate the features of the tool that enable us to specify and check system-specific properties of legacy code.

3.1 Expressive annotations

Our annotations are similar in spirit to those found in ESC-JAVA [15] for JAVA programs, and SPEC# [5] for C# programs, but are designed for verifying systems programs written in C. We provide an overview of the subset of

annotations that were used in this work. More details of the annotation language and advanced features are described in the HAVOC user manual available with the tool download[2].

Procedure contracts. Procedure annotations consists of preconditions, postconditions and modifies clauses. The `__requires` annotation specifies a precondition that holds at the entry to a procedure. This assertion is assumed when analyzing the body of the procedure and checked at all call-sites of the procedure. The `__ensures` annotation specifies a postcondition that holds at exit from the procedure. The `__modifies` annotation specifies a set of locations that are possibly modified by the procedure; it generates a postcondition that *all other* locations in the heap remain unchanged. The postconditions are checked when analyzing the body of the procedure, and assumed at all call-sites for the procedure.

Loop invariants. The `__loop_invariant` annotation specifies an assertion that holds every time control reaches the head of the loop. The assertion is checked at entry to the loop, and should be preserved across an arbitrary iteration of the loop.

Type invariants. Figure 4 illustrates type invariants for the `NODEA` and `NODEB` types, specified using a `__type_invariant` annotation. Type invariants specify assertions that hold for all objects of a given type. Such invariants typically hold at all control locations except a handful of procedures where an object is being initialized or being torn down, or may be broken locally inside a basic block (e.g. when an `NODEB` object is added as a child for `NODEA`). The user has the flexibility to choose the control locations where he or she expects the invariants to hold.

Proving type invariants require quantifier reasoning. Since reasoning with quantifiers is undecidable and highly unpredictable in practice [18], we provide an option to allow the user to assume the type invariants when checking other high level properties. These assumptions can be proved, but may require expensive analysis and significant annotation overhead.

Annotation expressions. A novel feature of our annotation language is that it allows most call-free and side-effect free C expressions in the assertions. The assertions can refer to user defined macros, thereby allowing complex assertions to be constructed from simpler ones. We allow referring to the return value of a procedure with the `__return` keyword. The postconditions may also refer to the state at the entry to the procedure using the `__old` keyword as follows:

```
__ensures  (__return == __old(*x) + 1)
__modifies (x)
int Foo (int *x) {*x = *x + 1; return *x;}
```

Resources. In addition to the C program expressions, we allow the annotations to refer to "ghost fields" (called *resources*) of objects. Resources are auxiliary fields in data structures meant only for the purpose of specification and manipulated exclusively through annotations. We allow the user to use `__resource ("NODEA_RESOURCE", NodeA)` to refer to the value of the `Resource` field in `NodeA`. The annotation `__modifies_resource (<NAME>, expr)` specifies that the resource `<NAME>` is possibly modified at `expr`. Consider the following annotation on the procedure `CompReleaseNodeA` that releases the `Resource` field of a `NODEA` object:

```
#define resrA(x) __resource(''NODEA_RESOURCE'', x)

#define __releasesA(x)\
        __requires (resrA(x) > 0)\
        __ensures  (resrA(x) == __old(resrA(x)) - 1)\
        __modifies_resource(''NODEA_RESOURCE'', x)

__releasesA(NodeA)
void  CompReleaseNodeA (NODEA NodeA);
```

Annotations for exception handling. Microsoft C compiler supports *Structured Exception Handling* (SEH) to deal with flow of control due to software and hardware exceptions. In SEH, the program can use either `__try`/`__except` blocks to implement an exception handler, or `__try`/`__finally` blocks to deal with cleanup along both normal and exceptional paths.

```
__try{
   //guarded code
} __except (expr) {
   //exception handler
   //code
}
```

```
__try{
   //guarded code
} __finally{
   //termination code
}
```

HAVOC allows `__may_throw` (`condition`) annotations on procedures with the meaning that the procedure *does not* raise an exception if `condition` does not hold at entry to the procedure. This allows specifying `__may_throw` (`TRUE`) on one extreme to indicate that any call to the procedure may throw an exception, and `__may_throw` (`FALSE`) on the other extreme to indicate that the procedure *never* raises an exception.

3.2 Checker

In this section, we provide a brief overview of the checker for verifying an annotated procedure. Interested readers can find more details in other works [6]. The main enabling techniques in the checker are:

Accurate memory model for C. The first step in checking an annotated source is to define meaning to the annotations and the operations in the program. HAVOC provides a faithful operational semantics for C programs accounting for the low-level operations in systems code.

HAVOC treats every C pointer expression (including addresses of stack allocated variables, heap locations, and values stored in variables and the heap) uniformly as integers. The heap is modeled as a mutable map or an array

[2] Available at http://research.microsoft.com/projects/havoc/

`Mem` mapping integers to integers. A structure corresponds to a sequence of pointers and each field corresponds to a compile-time offset within the structure. Pointer dereference `*e` corresponds to a lookup of `Mem` at the address `e` and an update `*x = y` is translated as an update to `Mem` at address `x` with value `y`. Annotation expressions are translated in a similar fashion.

Given an annotated C program, HAVOC uses the memory model to translate the annotated source into a a BOOGIEPL [11] program, a simple intermediate language with precise operational semantics and support for annotations. The resulting program consists of scalars and maps, and all the complexities of C (pointer arithmetic, & operations, casts etc.) have been compiled away at this stage. Example of the translation can be found in earlier work [6].

Generating precise verification conditions. HAVOC uses the BOOGIE [5] verifier on the generated BOOGIEPL file to construct a logical formula called the *verification condition* (VC). Construction of the VC uses a variant of the *weakest liberal precondition* computation [4]. The VC is a formula whose unsatisfiability implies that the program does not go wrong by failing one of the assertions or the contracts. Moreover, HAVOC ensures that the VC generated for a loop-free and call-free program is unsatisfiable *if and only if* the program does not go wrong by failing any assertion or contract present in the code. This is in sharp contrast to most other static analysis tools that lose precision at merge points, or worse still at every instruction.

Scalable checking using SMT solvers. The satisfiability of the VC is checked using a state-of-the-art *Satisfiability Modulo Theories* (SMT) solver Z3 [10]. SMT solvers are extensions of the Boolean Satisfiability (SAT) solvers that handle different *logical theories* such as equality with uninterpreted functions, arithmetic and arrays. These solvers leverage the advances in SAT solving with powerful implementation of theory specific algorithms. These tools can scale to large verification conditions by leveraging conflict-driven learning, smart backtracking and efficient theory reasoning. The modular analysis with efficient SMT solvers provides a scalable and relatively precise checker for realistic procedures up to a few thousand lines large.

4 Interprocedural inference

HAVOC, like any other procedure-modular checker, requires contracts to analyze each procedure. However, it can be extremely cumbersome for the user to annotate every procedure for a large code base such as COMP. Given a partially-annotated code base, where the user annotates the complex procedures, it is desirable to have an interprocedural analysis to infer simple annotations on the rest. In Figure 5, the procedure `CompCreateChildWithAttribute` creates a child of `NodeA` in `CompCreateNodeB` and then initializes different parts of the the child object and other data structures through several layers of deeply nested calls. Given the contract on `CompCreateNodeB`, we would like the tool to infer that `NodeB->ParentA->Resource` is held at entry to all the `CompInitializeX` procedures.

```
void CompCreateChildWithAttribute(
     PNODEA NodeA, ATTRIBUTE attr,...){

  PNODEB NodeB;
  CompAcquireNodeAExcl(NodeA);
  CompCreateNodeB(NodeA, &NodeB,..);
  CompInitialize1(NodeB, attr,...);
  ...
}

__ensures((*PNodeB)->ParentA == NodeA)
void  CompCreateNodeB(
      PNODEA NodeA, PNODEB *PNodeB,..
   );

void CompInitialize1(
     PNODEB NodeB, ..){

  <modify ParentA, State fields in NodeB >
  CompInitialize2(NodeB, ...);
}

void CompInitialize2(PNODEB NodeB,..){
  <modify ParentA, State fields in NodeB>
  CompInitialize3(NodeB, ...);
}
```

Figure 5. Procedure calls chains

In this section, we describe an annotation assistant based on the Houdini [14] algorithm in ESC-JAVA. The algorithm takes as input a partially annotated module along with a finite set of *candidate contracts* for each procedure in the module, and outputs a subset of the candidates that cannot be refuted along any interprocedural path. The algorithm performs a fix-point computation where at each step a candidate annotation that can be refuted is removed and the resultant set is reanalyzed.

In the original implementation in ESC-JAVA, the Houdini algorithm was used with customized candidate annotations — namely for checking non-null assertions on parameters and return values. In our case, we provide directives to add candidate annotations to the procedures in the module by the following two mechanisms:

1. For assertions about global state of the program, we have an instrumentation directive to add a global contract to *every non-entry* procedure. This can be used to add an assertion that a particular global variable is non-null, or a lock in a global data structure is not held.

2. We also provide an *type-based instrumentation* of arguments and return values for non-entry procedures.

For instance, we can direct HAVOC to add a precondition that `x->ParentA->Resource` is held, where `x` is a parameter to the procedure of type `NODEB`.

Note that we don't allow candidate contracts on the entry procedures, because they don't have any callers. If the candidate contracts along with the user-defined contracts on such an entry procedure are inconsistent, then any contract would be provable vacuously. For all other procedures, such inconsistent contracts will be detected at the call-sites. In our implementation, we identify a procedure as non-entry (or internal) if at least one of its argument has a type that is internal to the module.

5 Application on COMP

In this section, we describe our experience with applying HAVOC on a core driver COMP from the Windows®operating system. For the sake of security, we keep the component and the names of the procedures anonymous. The component has around 300,000 lines of code, excluding the sources for the kernel procedures. There are more than 1500 procedures present in the module. The code for the component has evolved over almost two decades, and each new generation inherits a lot of the code from the previous versions. Some of the procedures in the module have up to 4,000 lines of code, signifying the complexity and the legacy nature of the code base. COMP also heavily employs the Microsoft *Structured Exception Handling* (SEH) mechanism for C/C++ to deal with flow of control due to exceptions, as discussed in Section 3.1. This drastically increases the number of control flow paths in the program, since in the worst case any procedure can raise an exception.

We first provide a brief description of the main data structures involved in the synchronization protocols in COMP. We will focus on four main types: `NODE` that is the root type which can contain multiple instances of `NODEA`, `NODEB` and `NODEC` types.

Each `NODE` has an `ERESOURCE` field `NodeResource` and a mutex `NodeMutex` for synchronization. The `ERESOURCE` structure implements a reader-writer lock in Windows that can be recursively acquired. The `NodeResource` acts as a global lock for access to any `NODEA`, `NODEB` and `NODEC` objects within a given `NODE` (i.e. it is sufficient to acquire this lock to access any field in the `NODEA`, `NODEB` and `NODEC` objects).

Each `NODEA` object has a list of `NODEB` children (as described in Section 2) and a list of `NODEC` children. Each `NODEA` has a `ERESOURCE` field `Resource` that protects most of its fields and the fields of its children `NODEB` and `NODEC` objects; each `NODEA` also has a mutex `NodeAMutex` that protects a set of other fields in each `NODEA` and its `NODEB` and `NODEC` children.

Each `NODEA` also has an integer field `ReferenceCount` that signifies the number of threads that have a handle on a particular `NODEA` object — a positive value of `ReferenceCount` on an `NODEA` object indicates that some thread has a handle on the object and therefore can't be freed.

There is a global list `ExclusiveNodeAList` of all the `NODEA` objects for which the `Resource` has been acquired. A call to the procedure `CompReleaseAllNodeAResources` releases the `Resource` field of any `NODEA` on the `ExclusiveNodeAList`.

We have checked the following properties whose violation can lead to serious bugs:

1. **Ref-count usage.** We checked that for every execution path, the increments and decrements of the `ReferenceCount` field of a `NODEA` object are balanced. Decrementing the count without first incrementing could lead to freeing objects in use and a net increment in this field would correspond to a resource leak, as the `NODEA` object will not be reclaimed.

2. **Lock usage.** We check for the violation of the locking protocol for the various locks in `NODE` and `NODEA` objects. For a mutex field, we check that the lock is acquired and released in alternation; for a reader-writer lock which can be acquired recursively, we check that each release is preceded by an acquire.

3. **Data race freedom.** This is roughly the property that we described in Section 2, except that we monitor reads and writes for the other fields in these objects too. Since the `NodeResource` in a `NODE` object acts a global lock, we need the `Resource` field in a `NODEA` object be held only when the global `NodeResource` lock is not held.

4. **Teardown race freedom.** We check for races between one thread freeing a `NODEA` object, and another thread accessing the same object. Any thread freeing a `NODEA` object must hold that `NODEA`'s `Resource` exclusive, hold the parent `NODE`'s `NodeMutex`, and ensure that `NODEA`'s `ReferenceCount` is zero. Conversely, any thread accessing a `NODEA` must *either* hold the `NODEA`'s `Resource` shared or exclusive, hold the parent `NODE`'s `NodeMutex`, or have incremented the `ReferenceCount` field. These rules ensure mutual exclusion between threads freeing and accessing `NODEA` objects, and any rule violation could lead to a teardown race. This is an example of a property that requires system-specific knowledge.

Figure 6 summarizes the annotation effort and the distribution of the 45 bugs found for the four properties listed

Annotations	LOC
Property	250
Manual	600
Inferred	3000
Total	3850

Property	# of bugs
Ref-count	14
Lock usage	12
Data races	13
Teardown	6
Total	45

Figure 6. Summary of annotations overhead and bugs.

above. The "Property" annotations are specifications written to describe the property and also to specify the behavior of kernel procedures. The "Manual" annotations correspond to procedure contracts, loop invariants and type invariants for this module. Finally, the "Inferred" annotations are a set of annotations that are automatically generated by the annotation assistant described in Section 4.

Currently, our checker runs on the annotated code for COMP, and generates 125 warnings over the approximately 1500 procedures in 93 minutes — this corresponds to roughly 3.7 seconds spent analyzing each procedure on average. Most of the runtime (roughly 70%) is spent in a non-optimized implementation for converting C programs into BOOGIEPL programs, which can be significantly improved. Further, each source file (roughly 60 of them in COMP) in the module can be analyzed separately, and hence the process can be easily parallelized to reduce the runtime.

Out of the 125 warnings, roughly one third of the warnings correspond to confirmed violations of the four properties listed above. This is a fairly low false positive rate, given that we have not invested in various domain-specific filters to suppress the unlikely bugs. Currently the rest of warnings are violations of procedure contracts, which require adding annotations on other fields of these structures, or devising a new template for annotations (e.g. checking which fields of an object are non-null). We are currently working on reducing these warnings.

In the following sections, we discuss details of a few bugs, the breakup of the manual annotations and the inferred annotations. We also discuss the sources of unsoundness in the verification.

5.1 Bugs found

In this section, we describe two representative bugs from the set of 45 violations to the different properties. An interesting nature of most of the bugs is that they appear along exceptional paths — paths where some procedure raises an exception. This suggests the maturity and well-tested nature of the code as well as the fact that HAVOC can find these subtle corner cases. Besides, some of these synchronization bugs are hard to reproduce in a dynamic setting; the developers of the codebase suspected a leak in the `ReferenceCount` field but had been unable to reproduce it.

```
...
__try{
  ...
  NodeA = CompCreateNodeA(Context, ..);

  if (!CompAcquireExclNodeA(Context, NodeA,
                  NULL, ACQUIRE_DONT_WAIT )) {

      NodeA->ReferenceCount += 1;
      ...
      CompAcquireExclNodeA(Context, NodeA,
             NULL, 0 );
      ...
      NodeA->ReferenceCount -= 1;
  }
  ...
} __finally {
  ...
}
...
```

Figure 7. Reference count leak.

Reference count leak. Figure 7 illustrates an example of a bug that leads to a violation of the Ref-count usage property. In the example, an object `NodeA` of type `NODEA` is created in `CompCreateNodeA` and then an attempt is made to acquire the `Resource` in `NodeA` using the procedure `CompAcquireExclNodeA`. The procedure `CompAcquireExclNodeA` has the behavior that it can return immediately or perform a blocking wait on the `Resource` depending on whether the flag `ACQUIRE_DONT_WAIT` is specified or not. Hence, if the first non-blocking acquire fails in the `if` statement, then it tries a blocking acquire. Before doing that, it increments the `ReferenceCount` field to indicate a handle on this `NODEA` object; the field is decremented once the `Resource` is acquired. However, if the procedure `CompAcquireExclNodeA` throws an exception, then the `_finally` block does not decrement the `ReferenceCount` field, and hence this `NODEA` object will always have a spurious handle and will never be reclaimed.

Data race. Figure 8 illustrates an example of data-race on the fields of `NODEA` object. The procedure first acquires the `Resource` lock of an object `NodeA` in the first `if` block. The fields of `NodeA` are modified in the `SetFlag` macro and in the `CompUpdateNodeAAndNodeB` procedure. The access in `SetFlag` is protected by the `Resource` lock. However, the procedure `CompPerformSomeTask` calls the procedure `CompReleaseAllNodeAResources` transitively with a deeply nested call chain, that might re-

```
...
if (!CompAcquireExclNodeA(Context, NodeA, NULL,
                          ACQUIRE_DONT_WAIT)) {
    ...
    CompAcquireExclNodeA(Context, NodeA, NULL, 0);
    ...
}

SetFlag(NodeA->NodeAState, NODEA_STATE_REPAIRED);
...
CompPerformSomeTask(Context, ...);
...
if (FlagOn( ChangeContext.Flags, ... )) {
    CompUpdateNodeAAndNodeB(Context, NodeA,
                        ChangeContext.Flags);
}
...
```

Figure 8. Data race on `NODEA` object.

lease the `Resource` lock in any `NODEA` object. This means that the `Resource` lock is not held at entry to `CompUpdateNodeAAndNodeB`, although the procedure expects this lock to be held at entry to modify the fields of `NodeA`.

5.2 Manual annotations

We classify the main source of manual annotations in this section. In addition to the aliasing constraints and type invariants described in Section 2, we also annotated a variety of interesting conditional specifications and loop invariants.

Conditional specifications. Consider the contract on the procedure `CompAcquireExclNodeA` that was present in the two bugs described in Section 5.1.

```
__acquire_nodeA_excl(NodeA,
         !__thrown && __return != FALSE)
__ensures(!FlagOn(Flags, ACQUIRE_DONT_WAIT) &&
          !__thrown
                ==> __return != FALSE)
BOOLEAN CompAcquireExclNodeA (PCONTEXT Context,
        PNODEA NodeA, PNODEB NodeB, ULONG Flags);
```

Here `__thrown` is an expression to indicate whether a procedure has a normal exit or an exceptional exit. The first annotation (an annotation macro composed of `__requires`, `__ensures` and `__modifies`) describes the condition under which the `Resource` field of `NodeA` parameter is acquired. The second annotation specifies that if `ACQUIRE_DONT_WAIT` flag is not set, and the procedure does not throw an exception, then the return value is never `FALSE`.

Loop invariants. We also specified loop invariants when the property being checked depends on the fields or variables being modified inside a loop. The procedure `CompClearChildState` in Figure 2 provides an example of such a loop invariant. But a more common form of loop invariant arises due to the following code pattern:

```
BOOLEAN CompTryAcquireNodeA(PNODEA NodeA,..)
{
    BOOLEAN AcquiredFlag = FALSE;
    ...
    __try{
       ...
       __loop_invariant(AcquiredFlag == FALSE)
       while (true) {
            CallMightRaise1();
            if (..){
               CompAcquireNodeAExcl(NodeA);
               AcquiredFlag = TRUE;
               CallMightRaise2();
               return TRUE;
            }
       }
    } __finally {
       ...
       if (AcquiredFlag)
           CompReleaseNodeA(NodeA);
       ...
       return FALSE;
    }
}
```

The callers of `CompTryAcquireNodeA` expect that the procedure acquires the resource of `NodeA` at normal exit. However, in the absence of the loop invariant, the checker would report a false warning where the `CompReleaseNodeA` tries to release a resource without first acquiring it. This happens because in the absence of the loop invariant, the checker will report a path where the value of `AcquiredFlag` is `TRUE` at the loop head, the procedure `CallMightRaise1` throws an exception and control reaches the `__finally` block.

5.3 Inferred annotations

HAVOC's automatic inference capability generated a majority of the simple annotations (around 3000 of them) and was crucial to the automation of the tool for such a complex codebase (i.e. only 600 manually written annotations on around 1500 functions analyzed by the tool).

Figure 9 summarizes the main classes of annotations that were generated using the automated inference mechanism.

1. **May throw:** This infers the set of procedures that may throw an exception by transitively calling into a kernel procedure that could raise an exception (e.g. `KeRaiseStatus` or `ExAllocatePoolWithTag`), without any exception handlers along the control flow path to catch the exception.

2. `NodeResource` **held:** This infers a set of procedures where the lock `NodeResource` on the global `NODE` object is held at entry to ensure data-race freedom.

Annotations type	# of inferred annot
May throw	914
`NodeResource` held	107
`NodeMutex` not held	674
`NODEA Resource` held	360
`NODEA Resource` release all	210
OUT parameter modified	271
Parameter flag set	331
Total	2867

Figure 9. Distribution of inferred annotations.

3. `NodeMutex` **not held:** This infers a set of procedures where the `NodeMutex` field of the global `NODE` is not held at entry. Since most procedures acquire and release this lock locally inside a procedure, this annotation is useful for proving that locks are not acquired twice.

4. `NODEA Resource` **held:** This infers that the `Resource` field for an `NODEA` parameter or the `Resource` field for the parent of an `NODEB` or `NODEC` object is held at entry to a set of procedures. This along with `NodeResource` ensures absence of data races.

5. `NODEA Resource` **release all:** This infers the set of procedures that could release the `Resource` field of any `NODEA` object by a transitive call to the procedure `CompReleaseAllNodeAResources`.

6. **OUT parameter modified:** This adds a `__modifies(x)` annotation for an out parameter `x` that is modified inside a procedure.

7. **Parameter flag set:** This infers a set of procedures where a certain field of a parameter is set on entry to the procedures. The parameter captures the state of computations that span multiple procedures and is threaded through the nested procedure calls. The parameter `Context` in Figures 7 and Figure 8 is an example of such a parameter.

5.4 Assumptions

HAVOC provides a set of options that allows the user to introduce a class of *explicit* assumptions into the verification, which can be enumerated and discharged later with more annotations or a separate analysis. This allows the user of the tool to control the degree of unsoundness in the verification, and to recover from them using more annotations. This is in contrast to most other static analysis tools that bake these assumptions into the analysis and there is no way to recover from them. There are three main sources of such assumptions in our current analysis: (1) field safety, (2) type invariants and (3) free modifies.

Field safety. We assume that the addresses of two different *word-type* fields (fields that are not nested structures or unions) can never alias, i.e. a write to `x->f` does not affect any constraint involving `y->g`, when `f` and `g` are distinct fields. This assumption is mostly maintained with the exception of cases where the program exploits *structural subtyping* whereby two structures with identical layout of types are considered equivalent, even though the field names might differ.

Type invariants. The type invariants are currently assumed at user-specified control locations, and help to reduce false positives significantly when dealing with unbounded set of objects. Although such assertions can be proved in HAVOC [18], it requires reasoning with quantifiers and the annotation overhead is fairly high.

Free modifies. Specifying sound modifies clauses can be challenging for large code bases such as COMP. Currently, our analysis provides sound modifies clauses for the fields that are present in the property description (e.g. the state of the locks in different objects, reference count fields), but is unsound with respect to most other fields.

Of the three options, we believe that both field safety and the type invariants hold for the module with very few exceptions and separate the proof of the high-level properties from the proofs of type-safety and type/shape invariants. However, the free modifies option has the potential to suppress some real errors, and we are working on removing this option.

6 Related Work

There is a rich literature on static analysis tools for finding various defects in software programs. We categorize these tools into the following broad categories:

Annotation-based checkers. Our tool is closely based on the principles of ESC-JAVA [15] tool for JAVA programs and SPEC# [5] tool for C# programs. The main difference lies in our focus on systems program written in C with support for low-level operations in both the source and the annotation language. Although these tools have been applied to real programs, we believe that they have not been applied successfully to a module of comparable size or complexity as COMP.

Dedicated property checkers. A majority of the numerous static analysis tools developed for systems software in the last decade fall in this category — we highlight only a representative sample for the different properties that scale to several thousand lines of code. Examples of data race checkers include RELAY [20], LOCKSMITH [19], `RacerX` [12]. CALYSTO [2] finds null dereference bugs in C programs by using SAT solvers. The ASTREÉ an-

alyzer [8] uses abstract interpretation [7] to prove the absence of certain runtime errors such as buffer overruns, integer overflows in embedded safety-critical software. Most of these tools do not require user annotations, use novel algorithms based on data-flow analysis, often with the intent of finding bugs at the cost of unsound assumptions. These techniques complement HAVOC's ability and can improve the automation of our tool by automatically inferring more annotations for particular domains.

Extensible property checkers. Tools such as SLAM [3], BLAST [17] and ESP [9] are examples of sotware model checkers that check a property by exhaustively analyzing models of C programs. Their property languages allow specifying simple state-machines over the typestate of objects, and can express simple lock usage properties. These tools are most suited for checking properties on global variables, and lose precision and soundness when dealing with low-level operations and objects in the heap. *Meta-level compilation* [13] provides compiler extensions to encode patterns of violations for system-specific properties in a state-machine language *metal*, which are checked at compile time. The technique finds serious errors in systems code, but does not attempt to maintain soundness or guarantees about the absence of such bugs. SATURN [1] uses a logic programming framework to specify static analysis, but can be less precise than HAVOC because it does not use annotations.

7 Conclusions

In this work, we have demonstrated the feasibility of our general-purpose annotation-based property checker for systems software, the challenges involved in scaling such an approach to large code bases with low false alarm rate, and the tradeoffs involved. In future work, we would like to study the benefits of the annotations on an evolving code, and challenges involved in keeping the annotations updated with the code changes.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, 2007.

[2] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification (CAV '07)*, LNCS 4590, 2007.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, 2001.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, 2005.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, 2005.

[6] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, LNCS 4424, 2007.

[7] P. Cousot and R. Cousot. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL '77)*. ACM Press, 1977.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In *European Symposium on Programming (ESOP '05)*, LNCS 3444, 2005.

[9] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, 2002.

[10] L. de Moura and N. Bjorner. Efficient Incremental E-matching for SMT Solvers. In *Conference on Automated Deduction (CADE '07)*, LNCS 4603, 2007.

[11] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[12] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[13] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating Systems Design And Implementation (OSDI '00)*, 2000.

[14] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe (FME '01)*, 2001.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, 2002.

[16] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *28th International Conference on Software Engineering (ICSE '06)*, 2006.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL '02)*, 2002.

[18] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL 08: Principles of Programming Languages*, 2008.

[19] P. Pratikakis, J. S. Foster, and M. W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Programming Language Design and Implementation (PLDI '06)*, 2006.

[20] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC/SIGSOFT Foundations of Software Engineering (FSE '07)*, 2007.